第7章 指针

7.4 指针数组

7.4.1 字符指针数组

复杂 C/C++ 程序里常用到指针的数组（以特定类型的指针作为元素的数组）。

例：需要一组字符串，常用字符指针数组索引它们。如软件中错误信息常用一组字符串表示。分散管理不便。可定义指针数组，指针分别指向输出信息串常量。

也可定义其他类型的指针数组，如指向整数或者其他类型的指针的数组，下面讨论以字符指针为例。

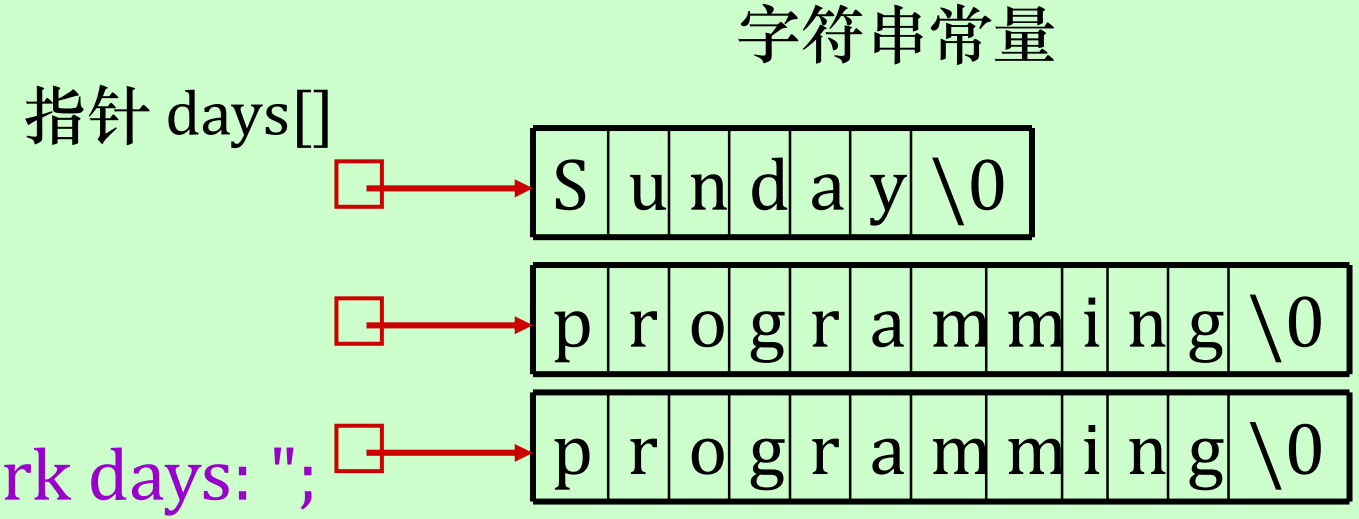
定义字符指针数组：

```
char *ps[10];
```

优先级也适用于定义。[] 优先级高，ps 是数组，其元素是字符指针。

定义字符指针数组时用字符串常量提供初始值。例：

```
char *days[] = { "Sunday",  "Monday", "Tuesday",
"Wednesday",  "Thursday", "Friday", "Saturday"
};
```



简单实例：

```
cout << "Work days: ";
for(i=1; i<6; ++i)
   cout << days[i] << "  ";
cout << "\nWeekend: ";
cout << days[6] << "  " << days[0];
```

字符指针数组实例：

改写第6章的C语言关键字统计程序，把原来的两维字符数组 keywords 改为字符指针数组。

只需定义下面数组，并用（关键字）字符串对各指针做初始化：

```
char *keywords[] = {
    "auto", "break",    ....
    ....     "volatile", "while"
};
```

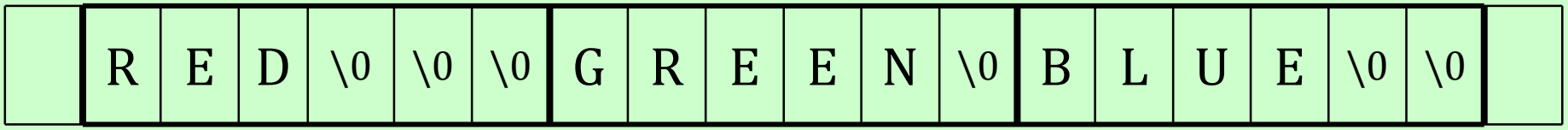
其他部分不需要改，程序可以正常工作。

7.4.2 指针数组与两维数组

两维字符数组与字符指针数组不同。定义：

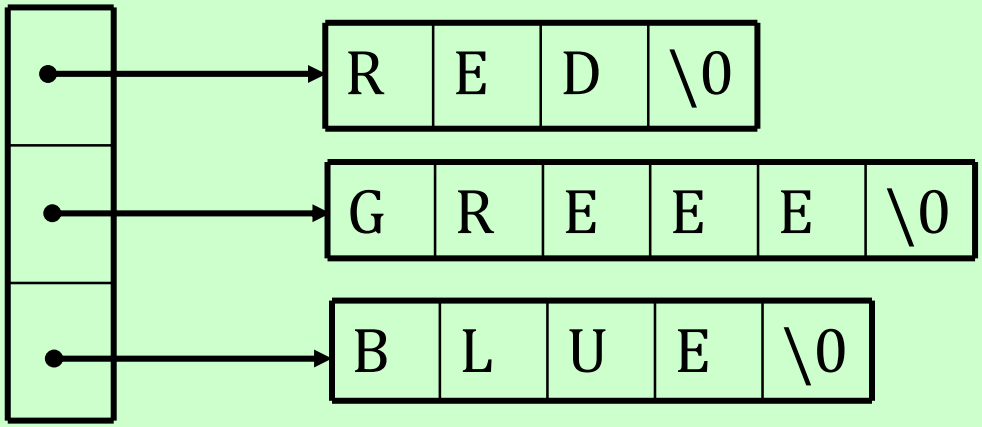
```
char color1[][6]={"RED","GREEN","BLUE"};
char *color[]={"RED","GREEN","BLUE"};
```

数组 color1



(a)

数组 color



(b)

*7.4.3 命令行参数的处理

启动程序的基本方式是输入命令，要求OS装入程序代码文件并执行。

命令行：描述命令的字符行。

在图形用户界面系统（如Windows）里，命令行存在于图标/菜单的定义中。

源文件 prog1.cpp 得到可执行文件 prog1.exe。

键入命令：

prog1

该程序就会被装入执行。

除命令名外，命令行常包括其他信息。DOS命令：

```
copy a:\file1.txt
dir \windows\system /p
```

附加信息也是字符序列，称为命令行参数。

前面的程序都没有包含处理命令行的功能

要写能够处理命令行参数的程序，需要用C语言的命令行参数机制。

处理命令行参数很像处理函数参数，写程序时要考虑和处理程序启动时实际命令行提供的信息。

命令行被看作空格分隔的字段，各个命令行参数。

命令名编号为0，其余参数依次编号。程序启动时把各命令行参数做成字符串，程序里可按规定方式使用。

设有程序 prog1；设启动程序的命令行是：

prog1 there are five arguments

这时prog1是编号为0的命令行参数，there 是编号1的命令行参数，…；共5个命令行参数。

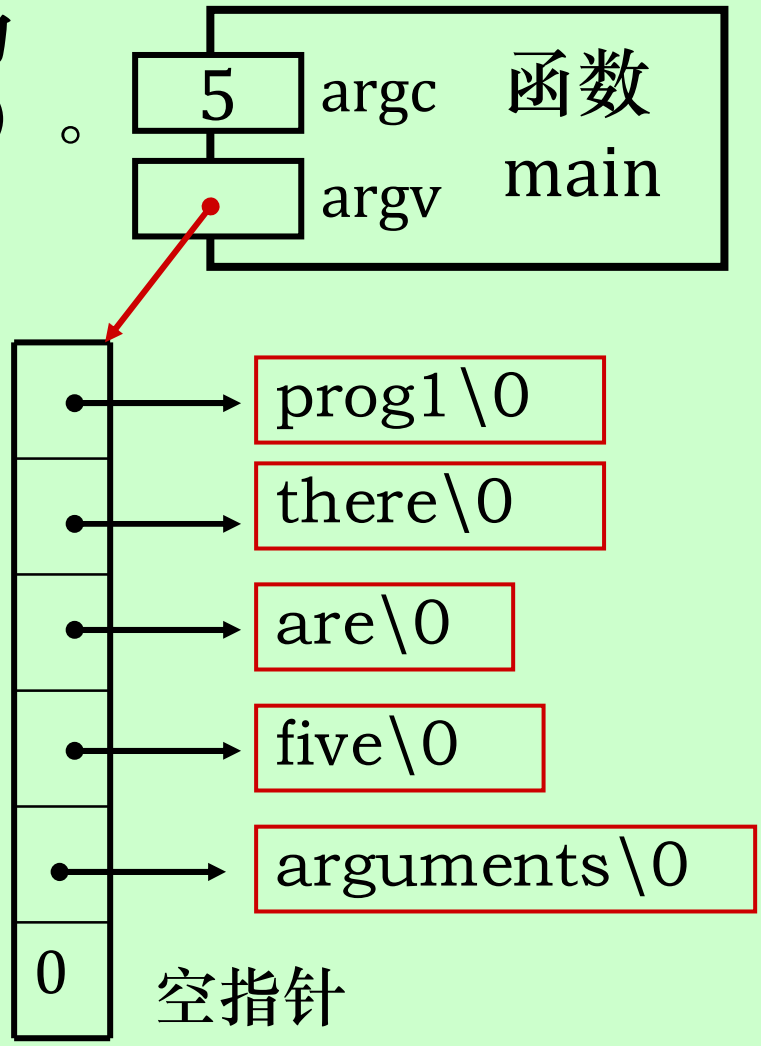
通过 main 的参数可获取命令行参数。main(void)表示不处理命令行参数，main的另一形式带两个参数：

int main (int argc, char *argv[]);

main 参数常用 argc、argv 作为名字（实际上可以用其他名字）。参数类型确定。

main 开始执行时：

- argc 是命令行参数的个数
- argv 指向含 argc+1个指针字符指针数组，前 argc 个指针指向各命令行参数串，最后有一个空指针



可由 argc 得到参数个数，通过 argv 访问它们。

可以访问启动程序的命令名本身。在一些系统里，0 号参数还包括完整的目录路径。

例：写程序 echo 打印各命令行参数。写程序时不知道调用时的命令行参数是什么，但可以打印它们：

```
#include <stdio.h>

int main (int argc, char *argv[]) {
    int i;
    for (i = 0; i < argc; ++i)
        printf("Args[%d]: %s\n", i, argv[i]);
    return 0;
}
```

书上有另一种定义方式，其中利用了最后的空指针

用 IDE开发程序时，编辑/调试/执行等工作都在环境里完成，执行程序时如何提供命令行参数?

集成开发环境都有专门机制为启动命令行提供参数。（如 Dev-C++ 的 运行 -> 参数）。

可转到 IDE之外在命令行状态下启动程序。在图形用户界面系统里，有关命令行参数的讨论同样有效。

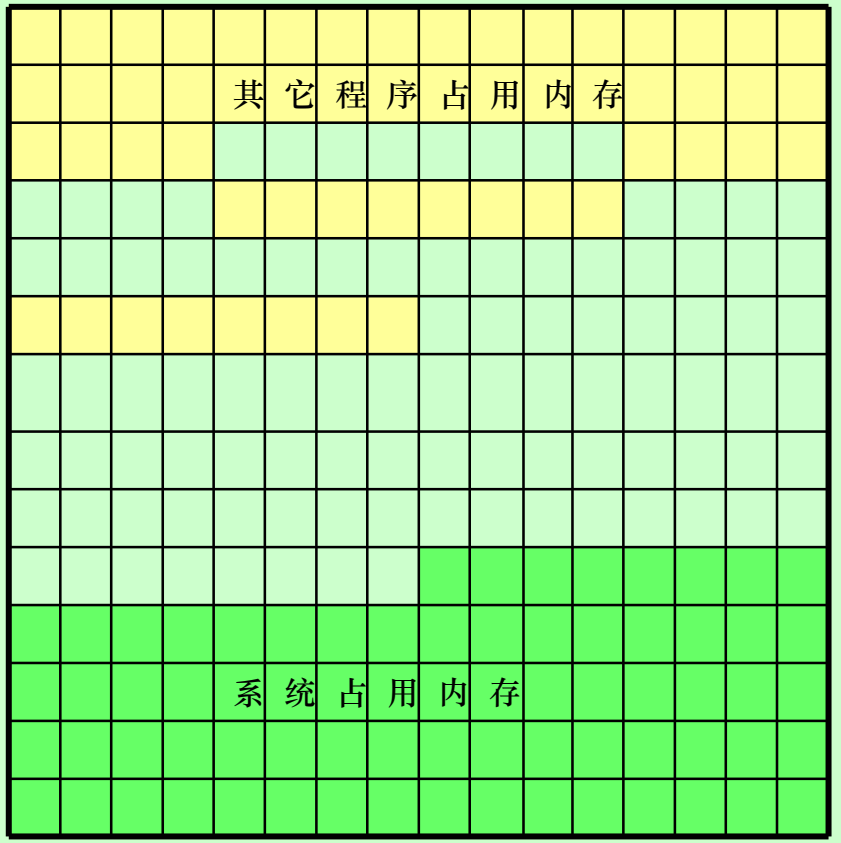
建立程序项、命令菜单项等也要写出实际命令行，包括提供必需的命令行参数。

一些图形界面系统里可把数据文件拖到程序文件上作为处理对象。此时将自动产生一个命令行。

第7章 指针

- 计算机系统含有内部存储器（内存）。计算机系统在工作时，把需要使用的程序和数据都加载到内存中。
- 每个字节都有一个隐含的唯一的地址。

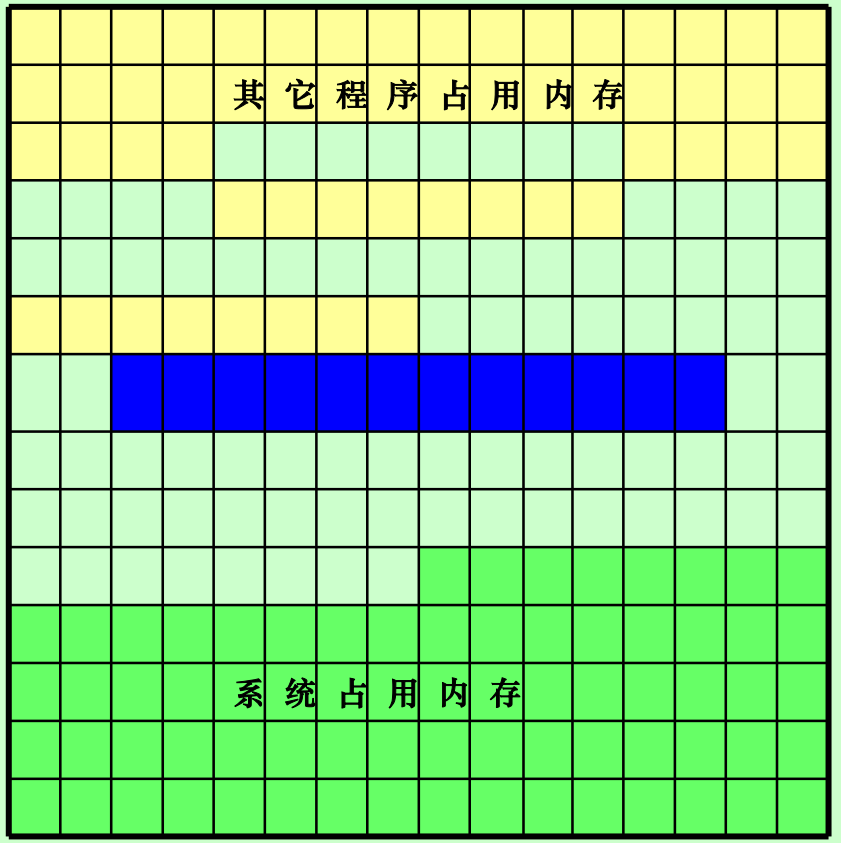
内存示意图：
（每格为一个字节）



系统所占用内存从一端开始，其它程序所用内存从另一端开始。

- 当程序开始运行时，系统自动给变量分配内存。
- 程序向系统申请动态分配内存时，系统会从空闲内存中挑选出连续的内存空间分配给该程序使用。
- 系统会记住“****-****这段内存已分配给**** 程序使用”

- 当程序对所动态申请的内存使用完毕时，应该主动通知“该段内存已使用完毕，请系统回收”。
- 如果不通知系统回收，则系统总是认为该段内存仍然处于被使用中。——可能导致内存用完。



7.5 动态存储管理

7.5.1 为什么需要动态存储管理

变量（简单变量/数组等）用于保存数据，需安排存储（称为存储分配）。

高级语言编程不需要考虑存储细节，有关工作由编译程序完成。编程效率高。

在 C /C++ 语言里

- 外部变量/局部静态变量在编译的时候确定存储，开始执行前分配存储
- 自动变量在执行进入定义函数时分配存储。

共同性质：变量大小都是静态确定的。

例：函数中变量和参数决定了函数执行时所需要存储空间量，C/C++ 语言要求自动数组的大小用静态表达式描述。这样，函数需要的存储量就可在编译时确定。

静态处理存储的优点是方便，效率高，执行中的工作简单，速度快。

但对编程方式加了限制，有些问题不好解决。

例：要处理学生成绩，需要用数组存放。但编程时并不知道运行时需要处理多少学生成绩，每次处理的成绩项数也可能不同。程序里预先定义的项数不能准确对应实际项数。

能否先通知数据项数，再建数据表示?

```
int n;
cin >> n;  //获得数据项数
double scores[n]; // ANSI C 不允许这样做!
... // 读入数据和处理
```

有时候事先根本不知道数据项数！上面方法也不行。

至今讨论的机制无法很好解决这类问题。

这里的问题：程序运行中需要使用存储，有时程序对存储的需求量在写程序时不能确定。

可能解决方案：

1）分析问题，定义适当大小的数组。若分析正确，一般都能处理。但数据很多时程序就不能用。

2）定义尽可能大的数组以满足任何需要。浪费大量存储资源。如有多个这种数组就更难办。系统可能无法容纳几个大数组，但实际上它们并不同时需要很大空间。

解决的办法是“动态存储分配”。在程序运行中做存储分配工作。

动态存储分配与释放

根据运行中的需要分配存储，取得存储块使用，称为动态存储分配。在运行中根据需要动态进行。

程序里怎样使用分配的存储块?

程序使用变量是通过名字。动态分配的存储块没有名字，因此需要其他访问途径。

借助于指针使用分配的存储块。用指针指向存储块，间接使用被指存储。访问动态分配存储是指针的最重要用途。

与此对应：动态释放，不用的动态存储块应交还。

动态分配/释放由动态存储管理系统完成，这是程序运行系统的子系统，管理着称作堆（英文heap）的存储区。大部分常规语言都有这种机制。

7.5.2 动态存储管理机制

C 和C++ 语言都具有完善的动态存储管理机制。两者的用法有所不同。

C++:

- new 运算符：动态申请存储空间。
- delete 运算符：释放由 new 申请的存储空间。

C:

- 存储分配函数 malloc()
- 带计数和清 0 的存储分配函数 calloc
- 动态存储释放函数 free
- 分配调整函数 realloc

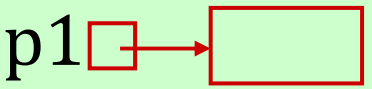
C++ 中的动态存储管理

new 运算符：动态申请存储空间。
delete 运算符：释放由 new 申请的存储空间。

1. 用 new 运算符申请单个变量的存储空间：

指针变量 = new 类型名;

申请一个 类型名 类型变量的空间，并返回该空间的起始地址。失败则返回 NULL 值。

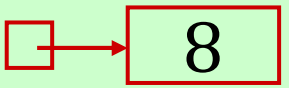


例：int *p1;　　p1 = new int;

*p1 = 8;　　//该空间只能通过指针 p 间接访问。

可以同时进行变量初始化：

p1 = new int(8);　//注意是用圆括号



2、用 new 命令申请数组的存储空间：

指针变量 = new 类型名 [表达式];

申请 表达式 个 类型名 类型变量的空间，并返回该空间的起始地址。失败则返回 NULL 值。

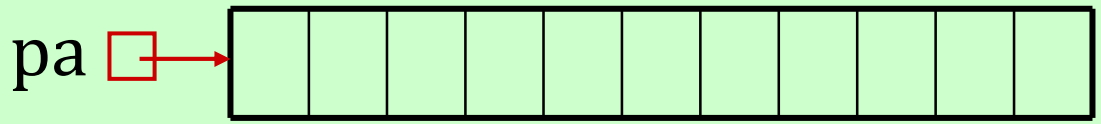
以后可以用该指针访问所申请到的数组存储空间。（可以用指针写法或数组写法）

例：int *pa;

pa = new int[10] ; //申请10个整数(即一维数组) 的空间，

... *(pa+i) ... ; //该空间只能通过指针 pa 间接访问。

... pa[i] ...;



申请多维数组是类似的。例如 int *q = new int [10][10];

★注意

使用 new 运算符动态申请空间，不是每次都能成功。

为保证程序执行正确，每次使用 new 申请空间后，都要测试是否成功（申请不成功时返回值为NULL）。

```
int *pa = new int[10] ;   //申请数组空间
if ( pa==NULL ) {  //或 pa==0，或 !pa
    cout << "动态分配出错！ \n";  exit(1);
    //出错时通常用 exit 函数结束程序运行，
    //返回预定义的错误代码。
 }
```

★由 new 动态申请的存储空间，
在程序结束前必须通过 delete 释放。

delete 运算符的两种格式：

格式一： delete 指针变量;

释放由 new 分配的简单类型变量的空间

例如：delete p1, p2;

格式二： delete [N]指针变量;

释放一个指针指向的数组空间。

N是常数，可省略。

例如：delete []pa;

[例] 使用动态数组

```
int  main( ) {
    int n, *pa, i;
    cin >> n ;
    pa = new int[n] ;   // 申请空间

    for(i=0; i<n; i++) // 使用空间
        cin >> pa[i];   // 或 cin >> *(pa+i);
    for(i=0; i<n; i++)
        cout << pa[i] << '\t';  // 或 cout << *(pa+i);
    cout << '\n';
    delete [] pa ;        // 释放空间
    return 0;
}
```

C语言的动态存储管理机制 memory allocation

用标准库函数实现，<stdlib.h> 或 <malloc.h>

1）存储分配函数malloc()。原型：

void *malloc(size_t n); /*size_t 是某整型类型*/

分配一块不小于n的存储，用通用指针返回其地址。

无法满足时返回空指针值。

```
int n; double *data;
... scanf("%d", &n);
data=(double*)malloc(n*sizeof(double));
if (data == NULL) {
  .... /* 分配未完成时的处理 */
}
..data[i]..*(data+j)../*正常处理*/
```

malloc 的返回值（void*）应通过类型强制转为特定指针类型后赋给指针变量。

使用注意事项：

- 分配存储块大小应该用 sizeof 计算
- 动态分配必须检查成功与否
- 动态分配的块大小也是确定的。越界使用（尤其是越界赋值）是严重错误，可能导致程序或系统垮台

2）带计数和清0的存储分配函数calloc。原型：

```
void *calloc(size_t n, size_t size);
```

size是元素大小，n是个数。

- 分配一块存储，足够存n个大小为size的元素，并把元素全部清0；无法分配时返回空指针值。

- 前面的存储分配问题也可用下面语句实现：

```
data = (double*)calloc(n, sizeof(double));
```

- 主要差别：malloc对所分配的区域不做任何事情，calloc对整个区域自动清0。

3）动态存储释放函数free。原型：

```
void free(void *p);
```

free 释放 p 指的存储块。注意：

- 该块必须是通过动态存储分配得到的
- p 值为空时什么也不做
- 执行free(p)后p值未变，被指块可能已变。不允许间接访问已释放存储块
- 不要对并非指向动态分配块的指针用本操作

为保证动态存储的有效使用，动态分配块不再用时应释放。动态存储块的释放只能通过调用free完成。

程序例子：

```
int fun (...) {
   int *p;
   ... p = (int *)malloc(...);
   ...
   free(p);  return ...;
} /*退出函数前应释放函数内分配且已无用的动态存储*/
```

fun退出时p存在期结束，若没有访问分配块的其他途径，将不可能再用到函数里分配的存储块。

动态存储的流失

如程序长期执行，存储流失就可能成为严重问题。对实际系统可能是很严重的问题。

4）分配调整函数 realloc。函数原型是：

```
void *realloc(void *p, size_t n);
```

更改已有分配。p指原分配块，n是新大小要求。

返回大小至少为n的存储块指针。新块与原块一致：新块小时保存原块n范围内数据；新块大时原数据存在，新增部分不初始化。分配成功后原块可能改变。

无法满足时返回空指针，原块不变。

常用写法(防止分配失败导致原存储块丢失)：

```
q = (double*)realloc(p, m * sizeof(double));
if (q == NULL) { /*未成功，p仍指原块，特殊处理*/ }
else {
  p = q; /* 令p指向新块，正常处理 */ …
}
```

7.5.3 动态存储分配程序实例

【7-7】把前文中的“筛法求素数”程序改为函数，然后写一个程序测试该函数：用户输入一个确定范围的整数值，调用筛法函数求出从2到该整数的素数，然后打印输出所有素数。

在这个程序里需要用数组存储一批整数。采用动态存储分配的方式：申请一个 int 动态数组。

为了让函数功能清晰、责任明确，应该让筛法函数只完成自己份内的功能，而把内存的动态分配和释放都留在 main 函数中：

筛法计算包装为函数：

```
void sieve(int lim, int an[]) {
  int i, j, upb = sqrt(lim+1);

  an[0] = an[1] = 0; // 建立初始向量
  for (i = 2; i <= lim; ++i) an[i] = 1;

  for (i = 2; i <= upb; ++i)
    if (an[i] == 1) // i是素数
      for (j = i*2; j <= lim; j += i)
        an[j] = 0; // i的倍数不是素数
}
```

```
int main() {
    int i, j, n=-1, *pn;
    do {  cout << "please input n (>=2): ";  cin >> n;
    } while (n < 2);

    pn = new int[n];
    if ( pn == NULL) { cout << "ERROR!\n";  exit(1); }

    sieve(n, pn);    //筛法函数

    for(j = 1, i = 2; i <= n; ++i)
       if (pn[i] == 1) {cout << i << (j%10 == 7 ? '\n' : ' '); ++j;}
    cout << "\n总个数: " << j <<endl;

   delete []pn;
    return 0;
}
```

动态分配内存

筛 法 求素数

动态释放内存

【例7-8】改造第6章的学生成绩统计和直方图生成程序，使之能处理任意个学生的成绩。

如何处理事先无法确定数目的数据集合。用数组限制了能处理的项数，现在改用动态分配。

让 readscores 根据需要申请存储块，返回动态分配的块和实际项数。函数原型：

```
double* readscores(int* np);
```

返回存储块的指针　　读取 *np 项数据

```
double* readscores(int* np) {
   int limit = 0, i = 0;
   double *tb;
   do{ cout << "请输入学生人数上限： " ;  cin >> limit;
   }while (limit <= 0);

   tb = new double[limit];   //!!! 动态分配存储空间
 //手工输入
   while (i < limit && cin>> tb[i])
      ++i;
   //从数据文件中读取
   //随机数模拟，分数区间为[30, 100]
    *np = i; //数据项数
   return tb;
}
```

```
int main() {
    int n;
    double *scores;
    if ((scores = readscores(&n)) == NULL)
        return 1;

    statistics(n, scores);
    histogram(n, scores, HISTOHIGH);

    delete []scores; //在 main 函数中要释放动态申请的存储

    return 0;
}
```

上面两个例子中的两种稍有不同的处理技术：

（1）在同一个函数里进行动态内存申请分配和释放；这种方法中的动态内存管理责任最为明确，不易出错。

（2）在一个函数中进行动态内存申请分配，并用函数返回值返回所申请得的动态存储空间的首地址。这种方法把动态存储的管理责任转移给了主调函数，主调函数必须要注意负责释放动态存储空间。

- 最好是使用第一种设计，因为它最清晰，也最不容易出现忘记释放的情况。

第7章 指针

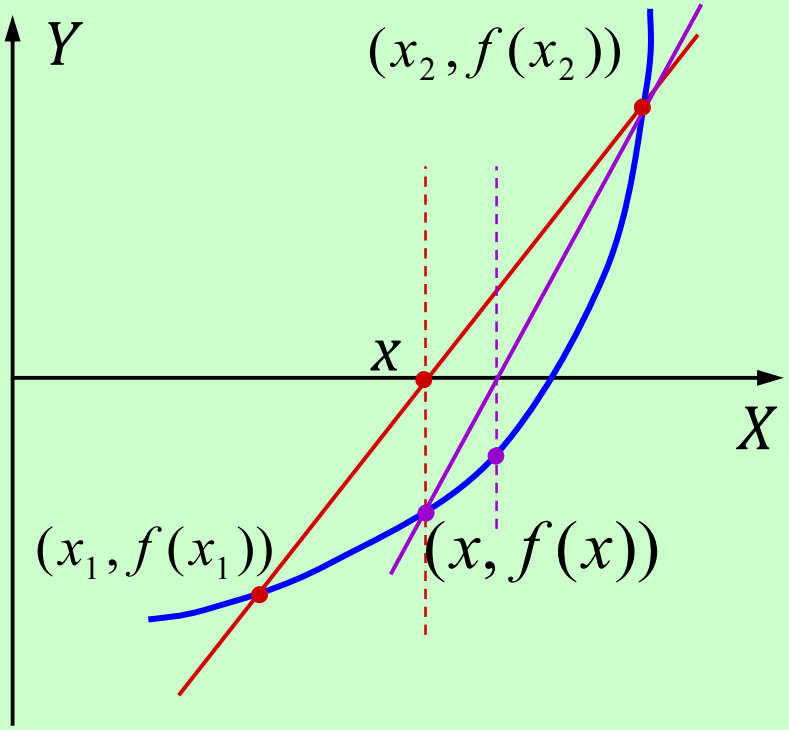
7.6.1 指向函数的指针

【例7-10】弦截法求函数根（典型数值计算问题）。

设有 函数 $y=f(x)$，求它与 X 轴的交点（根）。

过程：

1. 选定区间[x1, x2]，两端点函数值异号；
2. 做过端点弦线；求弦线与 X 坐标轴交点(x)；
3. 缩小区间，重复操作；直到交点函数值充分接近 0（满意为止）。



对程序进行的函数分解，考虑定义几个函数：

- 被求根数学函数是独立实体。原型说明：

  double f (double x);

- 求数学函数两端点的与坐标轴交点的公式：

$$x = \frac{x_1 \cdot f(x_2) - x_2 \cdot f(x_1)}{f(x_2) - f(x_1)}$$

  这是独立工作，定义函数 crossp，以端点坐标为参数，计算弦线与坐标轴的交点。

  double crossp (double x1, double x2);

- 求根计算定义为独立函数，可用在任何程序里，只要有被求根函数，以端点作为参数调用。

```
#include <iostream>
#include <cmath>
using namespace std;

double f (double x)   {
    return x * sin(x) - 2 * x * x + 2 * x;    //示例数学函数1
    //return ((x-5.0)*x+16.0)*x-80.0;    //示例数学函数2
}

double crossp (double x1, double x2) {
    double y1 = f (x1),  y2 = f (x2);
    return (x1*y2 - x2*y1) / (y2 - y1);
}
```

$$f(x) = x\sin x - 2x^2 + 2x$$

$$f(x) = \big((x-5)x+16\big)x-80$$

$$x = \frac{x_1 \cdot f(x_2) - x_2 \cdot f(x_1)}{f(x_2) - f(x_1)}$$

```
double chordroot (double x1, double x2) {
    double x, y, y1 = f (x1);
    do {
        x = crossp(x1, x2);     y = f (x);
        if (y * y1 > 0) { // y与y1同符号，新区间[x,x2]
            x1 = x;  y1 = y;
        } else
            x2 = x;  /*异号，新区间[x1,x]*/
    } while (fabs(y) >= 1E-6);
    return x;
}
```

```
int main() {
    double x1 = 2, x2 = 6;
    double x = chordroot (x1, x2);
    cout<<"A root of equation: "<< x <<endl;
    return 0;
}
```

- 分析上面的程序，可以注意到：被求根的数学函数写在函数 f 中，函数 cross 和 root 都调用函数 f 来求根。
- 如果要对其它数学函数求根，则需要重新改写函数 f 的函数体。或者写出 f1、f2 函数，然后编写出另一套 cross 和root 函数……
- 然而，求根解法是一个通用方法！希望能用它们处理多个数学函数！

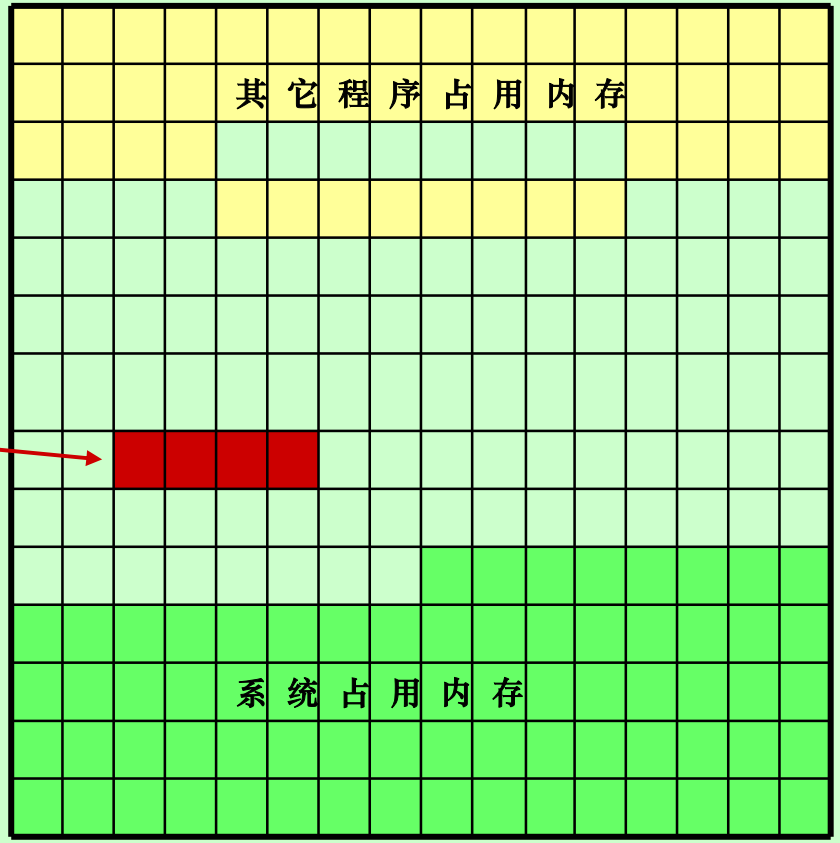
?

- 提高函数通用性方法：引进新参数。
- 要使求根函数能处理不同的数学函数，必须为它引进与函数有关的参数。

C和/C++ 语言里不能直接把函数作为函数的参数，而需要使用指向函数的指针（函数指针）。

在函数调用时，可以通过这种指针把所需函数传进去，从而达到在不同调用中使用不同函数的目的。

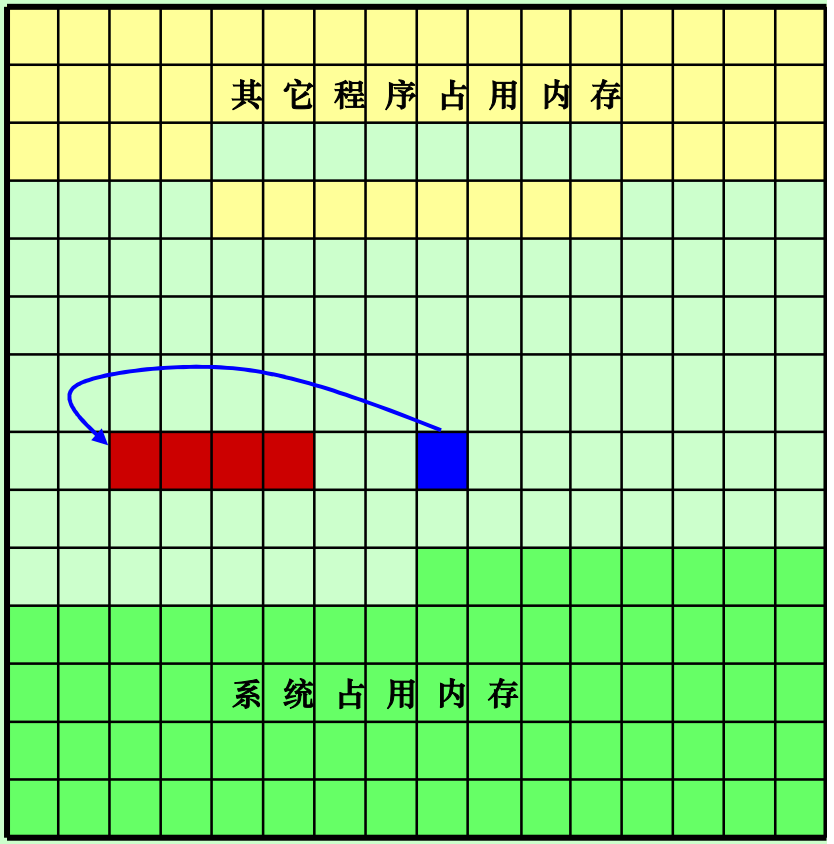
一个函数在执行时占用一段连续的内存区，而函数名就是该函数所占内存区的首地址。



可以把函数的这个首地址(或称入口地址)赋给一个指针变量，使该指针变量指向该函数。

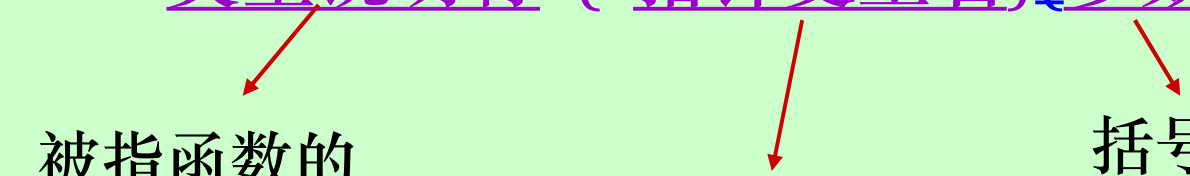
然后通过指针变量就可以找到并调用这个函数。

这种指向函数的指针变量被称为“函数指针变量”。



函数指针变量定义的一般形式为：

类型说明符 (*指针变量名)(参数表);



例如：double (*pf)(double, double);

函数指针的一个重要用途就是作为函数参数，通过函数指针使用函数，执行中使用的是哪个函数，就要看指针当时的值了。

采用函数指针的方式带来了新的灵活性。

以函数指针作为求根函数的参数，重新定义弦线法求函数根的函数：

```
double cross (double (*pf)(double), double x1, double x2) {
   double y1 = pf(x1), y2 = pf(x2);
   return (x1 * y2 - x2 * y1) / (y2 - y1);
}

double chordroot (double (*pf)(double), double x1, double x2) {
   double x, y, y1 = pf(x1);
   do {
      x = cross(pf, x1, x2);  y = pf(x);
      if (y * y1 > 0.0) { y1 = y;  x1 = x;
      } else    x2 = x;
   } while (y >= 1E-6 || y <= -1E-6);
   return x;
}
```

函数使用实例：

```
y = chordroot (f1, 1.2, 7.3);
y = chordroot (f2, 1.2, 7.0);
y = chordroot (sin, 0.4, 4.5);
y = chordroot (cos, 0.4, 4.5);
```

7.6.2 数值积分函数

【例7-10】写一个通用的数值定积分(numerical integration)函数。

这是使用函数指针参数的另一个例子，同时也讨论如何实现这种在数值计算中常用的计算过程。

$$\int_{x_1}^{x_2} f(x)$$

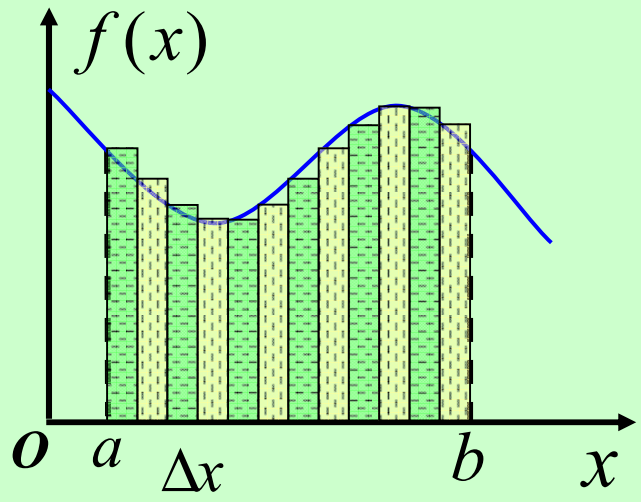
这个函数应有3个参数：对应被积函数的一个函数指针参数，表示积分限的两个double，返回双精度值：

```
double numInt(double (*pf)(double), double a, double b);
```

用区域分割法逼近积分值（矩形法 / 梯形法等）。

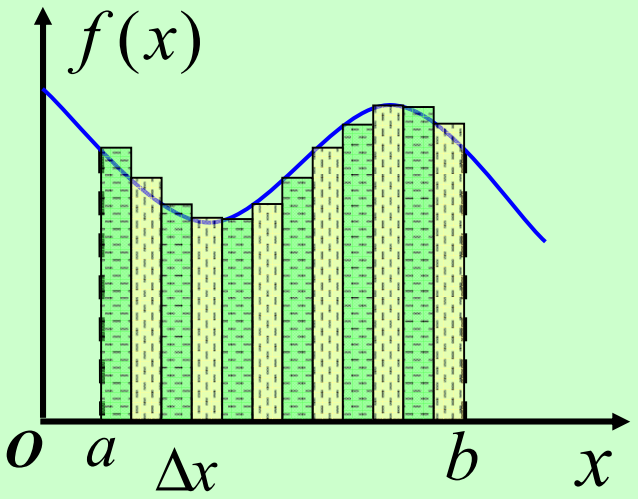
分割长度趋于 0 时有共同极限：

$$= \lim_{\substack{\Delta x \to 0 \\ n \to \infty}} \sum_{i=1}^{n} f(x_i)\Delta x$$



为简单起见，对积分区间采用等长划分和矩形方法，对每个区间用左端点计算：

```
double numInt(double (*pf)(double), double a, double b) {
    const int DIVN= 30;
    double res = 0.0, step = (b - a) / DIVN;
    for (int i = 0; i < DIVN; ++i)
        res += fp(a + i * step) * step;
    return res;
}
```



numInt不完善。

- 数学函数千差万别，统一划分方式不能满足各种情况。
- 可以通过增加划分提高结果精度（例如将划分数作为参数），但使用者很难确定合适的划分数。

一种方法：多次计算积分值。逐次加细划分再计算。若函数可积，这一系列结果将逐渐逼近实际积分值。

反复计算不能无限进行下去。合理的处理方法是在两次结果很接近时结束。

下面以两次积分值的差小于10^{-6}为结束条件。

```
double numInt(double (*pf)(double), double a, double b) {
    long i, divn = 10;

    double step, dif, res0,  res = (fp(b) + fp(a)) * (b - a) / 2;
    for (dif = 1.0; dif > 1E-6 || dif < -1E-6; divn *= 2) {
        res0 = res;
        step = (b - a) / divn;
        for (res = 0.0, i = 0; i < divn; ++i)
            res += fp(a + i * step) * step;
        dif = res - res0;
    }
    return res;
}
```

请考虑：这个函数能应付各种数学函数的积分吗？什么情况可能出问题？出什么问题？这些问题有解决方案吗？其中有没有很困难，以至根本无法解决的问题？

7.6.3 遍历数组

在编写与数组有关的程序中，读者可能已经注意到，程序中经常要对数组中所有元素依次进行一次某种操作。

例如全部依次赋予某个初值、全部依次进行某个数值变换、全部依次进行打印输出等。

可以稍为抽象地说，这是对数组进行一次遍历（Traversal)：沿着某种路线，依次对数组中每个元素均做一次且仅做一次访问（visit）。访问是指对数组元素的某种处理（赋值、打印或其它）。

【例7-11】先写一个对实数数据进行格式化打印（固定宽度，每5个换行）的函数 prt5(double x)，然后写一个对实数数组遍历的函数 traverse(int len, double *array, int (*visit)(double))，最后写一个主函数调用遍历函数对一个示例数组进行遍历（打印输出）。

```
int prt5(double x) {
    static int k = 0;
    cout << fixed << x << (++k %5 == 0 ? "\n" : "\t");
    return 0;
}

int traverse(int len, double *array, int (*visit)(double)) {
    for (int i = 0; i < len; i++)
        visit(array[i]);
}

int main() {
    const int LEN=20;
    double arr[LEN];
    for (int i = 0; i < LEN; i++)  arr[i] = sin(i);
    traverse(LEN, arr, prt5);
    return 0;
}
```

这个程序示例相当简单，
但是这个程序框架很有用。

本章小结

1. 指针是C语言中的重要功能，使用指针编程有以下优点：

① 提高程序的编译效率和执行速度。

② 通过指针可使用主调函数和被调函数之间共享变量或数据结构，便于实现双向数据通讯。

③ 可以实现动态的存储分配。

④ 便于表示各种数据结构，编写高质量的程序。

2. 指针的运算

① 取地址运算符 &：求变量的地址

② 取内容运算符 *：表示指针所指的变量

③ 赋值运算

- 把变量地址赋予指针变量
- 同类型指针变量相互赋值
- 把数组、字符串的元素地址赋予指针变量
- 把函数入口地址赋予指针变量

➢ 本章小结

④加减运算

对指向数组或字符串的指针变量可以进行加减运算，如p+n，p–n，p++，p--等。对指向同一数组的两个指针变量可以相减。

对指向其它类型的指针变量作加减运算是无意义的。

⑤关系运算

指向同一数组的两个指针变量之间可以进行大于、小于、等于比较运算。指针可与0比较，p == 0表示p为空指针。

➢ 本章小结

3. 与指针有关的各种说明和意义见下表

定义	含义
int i;	定义整型变量i
int *p;	p为指向整型数据的指针变量
int a[n];	定义含n个元素的整型数组a
int *p[n];	n个指向整型数据的指针变量组成的指针数组p
int (*p)[n];	p为指向含n个元素的一维整型数组的指针变量

例如，下列定义的含义：

```
（1）int *p[3];          //指针数组
（2）int (*p)[3];        //指向一维数组的指针
（3）int *p (int);       //返回指针的函数
（4）int (*p)(int);      //指向函数的指针，函数返回int型变量
（5）int *(*p)(int);     //指向函数的指针，函数返回int型指针
（6）int (*p[3])(int);   //函数指针数组，函数返回int型变量
（7）int *(*p[3])(int)   //函数指针数组，函数返回int型指针
```

➢ 本章小结

4. 指针常常与数组、函数联系在一起

5. 关于括号

在解释组合说明符时，标识符右边的方括号和圆括号优先于标识符左边的“*”号，而方括号和圆括号以相同的优先级从左到右结合。但可以用圆括号改变约定的结合顺序。

6. 阅读组合说明符的规则是“从里向外”。

从标识符开始，先看它右边有无方括号或园括号，如有则先作出解释，再看左边有无*号。如果在任何时候遇到了闭括号，则在继续之前必须用相同的规则处理括号内的内容。

例如： int *(*(*a)( ))[10]

上面给出了由内向外的阅读顺序，下面来解释它：

① 标识符a被说明为；
② 一个指针变量，它指向；
③ 一个函数，它返回；
④ 一个指针，该指针指向；
⑤ 一个有10个元素的数组，其类型为；
⑥ 指针型，它指向；
⑦ int型数据。

- 因此 a 是一个函数指针变量，该函数返回的一个指针值又指向一个指针数组，该指针数组的元素指向整型量。